Source Code Readability Improvement Using Heuristic-Based Dynamic Error Reporting During Editing

Phillip Anthony Relf

Doctor of Philosophy in Engineering

2007

CERTIFICATE OF AUTHORSHIP/ORIGINALITY

I certify that the work in this thesis has not previously been submitted for a degree nor has it been submitted as part of requirements for a degree except as fully acknowledged within the text.

I also certify that the thesis has been written by me. Any help that I have received in my research work and the preparation of the thesis itself has been acknowledged. In addition, I certify that all information sources and literature used are indicated in the thesis.

Signature of Student

_____

Acknowledgement

I thank my supervisors David Lowe and John Leaney for their patient direction and pertinent review comments, and particularly early on for their restraint when I was struggling to understand the full implications and the nature of the research process. I owe these two gentlemen much for their gentle direction in steering my paradigm shifts that have occurred during the prior years. In addition, I thank the following people for their support during my research: Susan Cockshell for reviewing and offering advice on questionnaire layout and wording, and also for Human Factors advice relating to the instructions for the experiment test subjects; Chris Coreless for removing the Java syntax errors from my translation of the original Ada experiment source code files into the Java computer language; the 2005 UTS classes: Software Systems Analysis, Software Systems Design and Software Analysis & Design, and also to Mark Denford and Jared Berghold for their participation as experiment test subjects. I also acknowledge the participation of the many corporate programmers who responded to questionnaires and supplied data from their participation in the experiment, and in particular to Anthony Anger, Chris Coreless, Martin Fowel, Grant Hensley, Ken Lau, Patrick Lee, Sun-Young Kim and Amir Sadross for their participation in the case studies. Finally, I thank my employer for allowing me to collect and report data as part of the industrial case studies.

I thank those anonymous reviewers who supplied comment on my research papers.

I wish to again acknowledge David Lowe for supplying editorial input.

For my parents: Dorothy, Gertrude, Ron and Sid.

Table of Contents

List of Tables

List of Figures

Abstract

This research considers whether dynamically reporting poor identifier-naming practices at the time when the source code is written can improve readability and hence maintainability. Poor identifier-naming practices have little effect on the production phase of the software lifecycle. However, poor identifier-naming practices can have a substantial impact during the maintenance phase of the software lifecycle, particularly for the maintenance of large (i.e., 1M SLOC) computer programs. Of the nineteen identifier-naming style guidelines employed to support the research and used to identify poor identifier-naming practices, thirteen were found to be useful in improving source code readability. A questionnaire was employed to ascertain whether expert programmers accepted these guidelines; a textbook survey was used to identify the potential to transmit poor identifier-naming practices; a survey of contemporary source code was used to ascertain current identifier-naming practices; and a survey of dated source code was used to ascertain how identifier-naming practices have changed over an extended period of time. In addition, a controlled experiment was used to evaluate the effects of poor identifier-naming during a maintenance exercise and to evaluate the generation of poor identifier-naming during a production activity. A novice programmer case study and a programming team case study were executed to identify the longer term effects of dynamically reporting poor identifier-naming practices. The benefit of dynamically reporting poor identifier-naming practices was most pronounced for novice programmers with the percentage of meaningful identifier names increasing from 12% to 28%. The results for expert programmers were less pronounced with the percentage of meaningful identifier names correspondingly increasing from 53% to 60%. The identifier-naming style guidelines that proved to be the most useful to programmers required that identifier names should be composed of from two to four Natural language words or project accepted acronyms; should not be composed only of abstract words; should not contain plural words; and should conform to the project naming conventions.

List of Abbreviations and Acronyms

ACM	Association for Computing Machinery
API	Application Programming Interface
CATS	Computer Architecture Topography Simulator
CD	Compact Disk
COCOMO	Constructive Cost Model
df	Degrees of Freedom
DoD	Department of Defense
Exp.	Experiment
GUI	Graphical User Interface
Id	Identifier
IDE	Interactive Development Environment
IEEE	Institute of Electrical and Electronic Engineers
Lab.	Laboratory
LOC	Line of Code
N/A	Not Applicable
PC	Personal Computer
ROM	Read Only Memory
SDD	Software Design Description
SLOC	Source lines of Code
SPC	Software Process Consortium
SRS	Software Requirements Specification
UML	Unified Modelling Language
USA	United States of America
UTS	University of Technology, Sydney
WYSIWYG	What You See Is What You Get
Y2k	Year 2000

Glossary of Terms

Most technical terms in this thesis are used consistently with the definitions given in IEEE Std 610.12-1990: *IEEE Standard Glossary of Software Engineering Terminology*. Additional terms used within the thesis, but which are not given in this standard are defined below.

Mission Critical

Mission Critical, when applied to a computer system, defines that computer system as being critical to the successful completion of the mission.

SLOC

The term SLOC is potentially ambiguous when used to identify the size of a computer program, in that the count of source lines of code can mean: (1) the total number of source lines of code generated during the production of the computer program; (2) the total number of source lines of code delivered to the customer; or (3) the total number of source lines of code necessary to build the computer program, excluding any support software. The last usage of the term SLOC hence excludes any SLOC counts attributed to test harness software and test program software. This last usage of the term SLOC has been assumed and is used within the body of this thesis.